How Sourcegraph's code AI platform powers Cody

Sourcegraph Whitepaper – June, 2023

Cody is an AI coding assistant for developers, answering technical questions, fixing and generating code with high accuracy, directly in their text editor or wherever they're writing and reading code. It is built on top of Sourcegraph's code AI platform, giving Cody the context and intelligence it needs to deliver a best-in-class AI assistant experience.

In this document you'll learn how Cody works and how exactly it makes use of Sourcegraph's AI platform. Through a close look at what goes on under the hood of a typical user interaction with Cody you'll learn how the platform gives Cody a unique advantage compared to other coding assistants today and how that advantage will grow in the future.

Cody & Sourcegraph's code AI platform

Cody, like other AI assistants, uses a Large Language Model (LLM) under the hood to answer a user's question. A big challenge when working with LLMs is that they don't have access to a user's data. Even though they've been trained on large amounts of data, they don't know about a user's codebase, their documentation, the currently open file in their editor, or other things the user might have questions about.

LLMs are context-free; they know of the world, but they don't know the user's world. LLMs can, for example, produce code in multiple programming languages. Through their training they've acquired the ability to speak languages, but they can't speak in the dialect spoken in a user's organization. The problem is that users want them to do exactly that: answer questions about *their* code, their organization, the problem they are currently facing.

One solution to that is to give context to the LLM: prepend a user's question with relevant information that might be necessary to answer it. If the user asks "is there a typo in this document?", the best way to get the LLM to answer truthfully is to add the document to the interaction.

In fact, different AI assistants already do that in different ways to solve the problem of missing context. ChatGPT, for example, allows users to paste all the necessary context into the chat window, amassing more and more context with each message. Most other AI coding assistants look at the file and directory the user currently has open in their editor and use that.

That is only solving half of the problem, though. The other half is finding the *right context* to use in a given interaction. If the user's codebase is millions of lines of code or they have 200 files open in their editor, which ones do you pick when the user asks how to fix a bug?

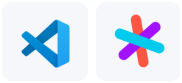
Cody uses Sourcegraph's code AI platform to answer that. It uses the platform to find out which type of context to add to a given interaction and then again uses the platform to gather that context from all

across a user's or organization's complete codebase - no matter how many million lines of code it is and where they're hosted.

Cody

Chatbot
Recipes
Auto-complete
Fixups

Available in:



Soon available in:

AI platform

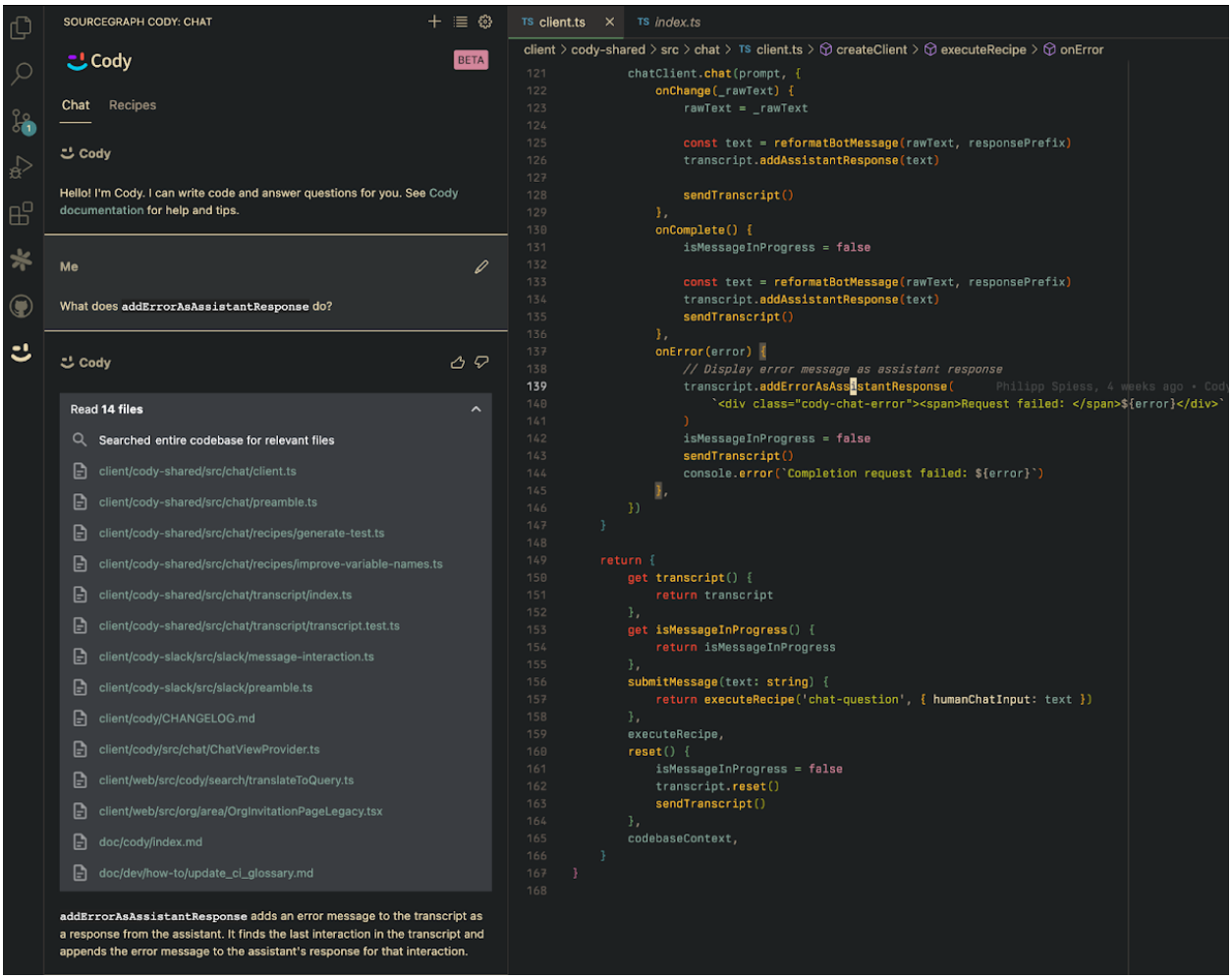
Code graph:
All your code & data

Language models:
Plug-n-play access to LLMs and embeddings models

Let's take a closer look at an interaction between a user and Cody to see what exactly is meant with Cody, code AI platform and how they interact.

Interaction between Cody and Sourcegraph's code AI platform

Here is a screenshot of a typical interaction a user might have with Cody inside their editor:



What you can see is that the user asked the following question about the code in the currently open file:

> *What does* `addErrorAsAssistantResponse` *do?*

The user refers to the invocation of `addErrorAsAssistantResponse` in the editor – implicitly, without pointing Cody directly at the line.

Cody's reply contains two things:

1. A list of files in the user's codebase that Cody read.
2. An answer to the question, explaining what the function does.

It's noteworthy that the answer not only explains what `addErrorAsAssistantResponse` does in the currently open file, but that it also explains what the function itself does ("It finds the last interaction…").

That information is contained in `transcript/index.ts`, which is in the list of files that Cody read. If we look at the definition of `addErrorAsAssistantResponse` we can see that this is where Cody got its information from:

```JavaScript
public addErrorAsAssistantResponse(errorText: string): void {
    const lastInteraction = this.getLastInteraction()
    if (!lastInteraction) {
        return
    }
    // If assistant has responsed before, we will add the error message after it
    const lastAssistantMessage = lastInteraction.getAssistantMessage().displayText || ''
    lastInteraction.setAssistantMessage({
        speaker: 'assistant',
        text: 'Failed to generate a response due to server error.',
        displayText:
            lastAssistantMessage + `<div class="cody-chat-error"><span>Request failed:
</span>${errorText}</div>`,
    })
}
```

In other words: Cody gave a high-quality, contextualized answer by taking the user's codebase (not just the currently open file) into account.

How?

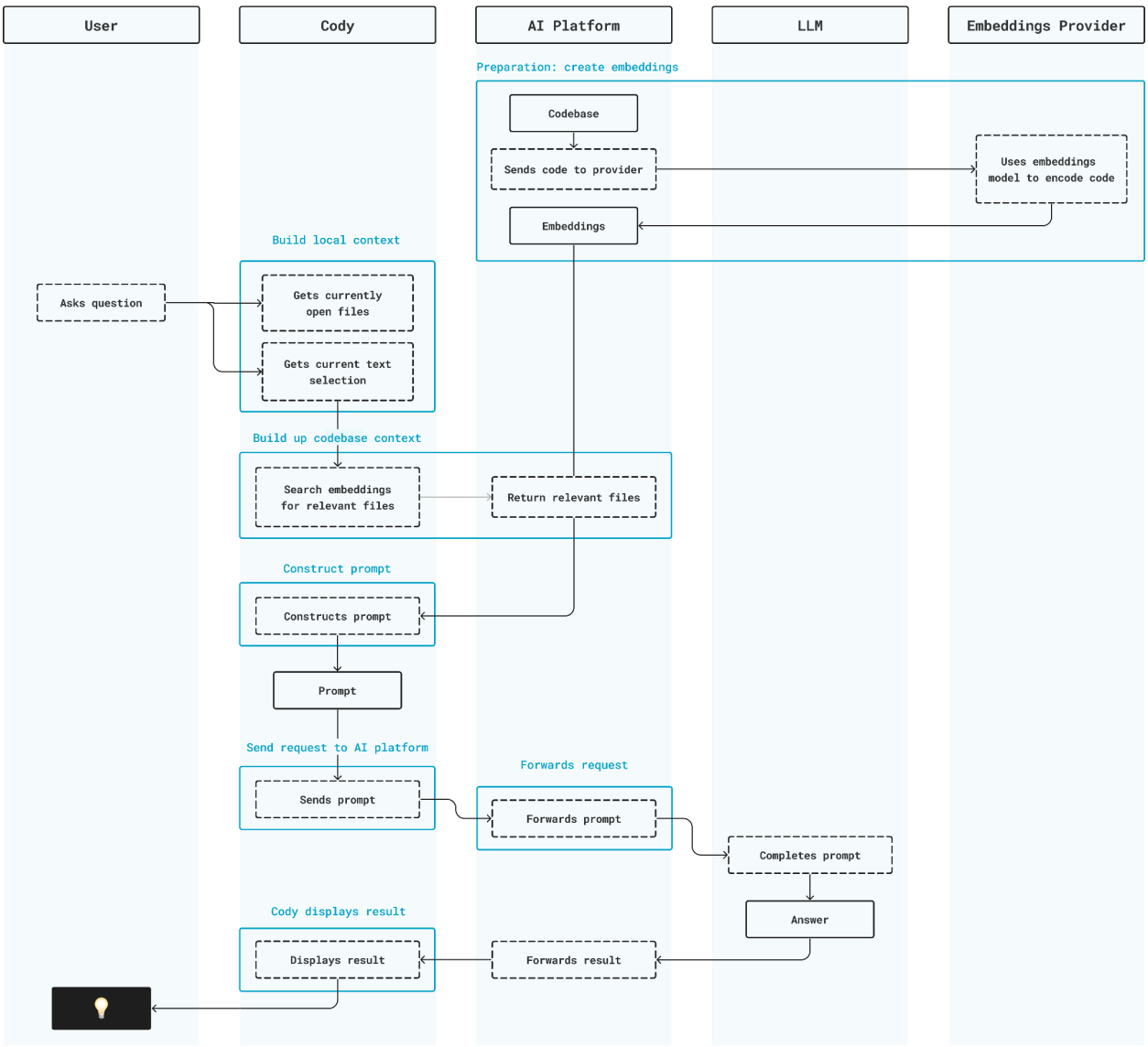


Diagram of a typical interaction between user, Cody, code AI platform, LLM, and embeddings provider.

1. Preparation: create embeddings

Before this interaction could take place, a few things had to happen:

1. The user's codebase was synced to the code AI platform the user is connected to (either sourcegraph.com or a custom instance).
2. Embeddings were created for the codebase.

The first step allows the code AI platform to serve the codebase and its code graph to Cody, the second allows Cody to find the most relevant files to a user's query.

The Cody white paper on context contains more information about embeddings, but for the purposes of this exploration let's gloss over some details and say that embeddings are representations of your code, effectively turning the code into data that captures context and semantic relationships.

The AI platform creates embeddings by sending a repository's contents to OpenAI's embeddings API, which, to simplify, takes in a list of text and returns a list of numbers (i.e. the encoding of code in vector space). These embeddings are then stored in the storage provider configured in the platform (for example: S3, GCS, or another supported blob store), from which they're retrieved and cached on demand.

In the future (see Cody & Sourcegraph's code AI platform in the future), the code AI platform will be able to create embeddings without the use of third parties, such as OpenAI.

By creating embeddings for a codebase, Sourcegraph's code AI platform allows Cody to find the files that are most likely to contain answers to a user's question ("how do we display timestamps in React components?") or clues on how to execute an instruction ("change this code to use our timestamp component").

2. Build up local context

With embeddings created and code synced to the code AI platform, Cody is ready to answer the user's question.

Cody's first task: gather context to send along with the question to the LLM. To do that, Cody builds up two types of context:

1. Local context, which is the context in the editor of the user. E.g.: the currently open file or text selection in the editor.
2. Codebase context, which is context gathered from across a user's complete codebase, achieved by querying the Sourcegraph code AI platform.

(In the future, Cody will be able to gather context from third-party data sources too: a company's internal wiki, documentation pages, tools, etc.)

Let's take a look at the local context first. Here is the relevant code that shows how Cody builds that up:

1. It checks whether there's a current text selection in the editor it should include. If so, it's included.
2. If the current chat question (or recipe) requires local context from the editor it adds the currently open file to the context.

The "if" in point 2 refers to something called "intent detection", which happens locally (in the user's IDE) for local context. In the future this might be moved to the code AI platform and replaced with a request, since there's many optimizations to be done that might require more sophisticated access to data than the environment of the user's IDE allows.

3. Build up codebase context, by searching embeddings

In order to determine whether the user's current question requires codebase context, Cody sends a request to the code AI platform that contains the user's latest question or instruction. Cody is asking the platform: is non-local context (meaning: context outside the user's editor) required to give a good reply?

On the platform side, the query is analyzed and, again, run through intent detection. If the query, for example, refers to a previous message in the conversation ("explain this in more detail"), the intent detector returns false to signal that codebase-wide context is not required. The assumption here is that the existing conversation already contains enough context. If the user writes "Hi! How's it going?" the intent detector also determines that no context is required (yet). On the other hand, if the query does sound like it requires context, the intent detector returns true..

If Cody gets a `true` back from the platform, it replies with another request, this time searching the embeddings that were produced in step 1. That request is another GraphQL request, this time using the `embeddingsSearch` query to find code and text (documentation & other text files) files that are likely to be relevant to the user's query.

This also means that Cody doesn't require searching through the embeddings if the intent detector determines that local context is enough to provide an answer – making the interaction faster and more efficient, since no additional roundtrip is required and less context needs to be sent to the LLM for it to return a helpful answer.

4. Construct prompt

At this point, Cody has enough context to start preparing its requests to the LLM:

1. Local files and snippets, if the local intent detector signaled they would be useful.
2. Codebase files and snippets, if the platform intent detector signaled they would be useful.

The next step: construct a prompt for the LLM.

The prompt Cody builds is composed of multiple messages, a dialog between `human` and `assistant`, in which the `human` (Cody, on behalf of the actual human – the user) shares a preamble, all the relevant context it gathered in the previous steps, and a question or instruction with the `assistant`.

This dialogue between `human` and `assistant` is then passed to the LLM, which completes the last line, adding what the `assistant` says based on everything that came before. The idea is to give the LLM a script and ask it: based on everything in this script, what would the next message of the `assistant` – you, the LLM – be?

Here is an example prompt, shortened, but still showing the essential pieces: the preamble (`"You are Cody, [...]`"), the context (`"Use the following text from file [...]"`), and, finally, the last undefined message by `assistant`, which is the LLM's job to fill out:

```
Unset
{speaker: 'human', text: 'You are Cody, an AI-powered coding assistant created by Sourcegraph. [...]'}
{speaker: 'assistant', text: 'Understood. I am Cody, an AI assistant made by Sourcegraph [...]'}
{speaker: 'human', text: 'Use the following text from file `CHANGELOG.md`:  [contents of
CHANGELOG.md]'}
{speaker: 'assistant', text: 'Ok.'}
{speaker: 'human', text: 'Use the following text from file `main.go`:  [main.go]'}
{speaker: 'assistant', text: 'Ok.'}
[...]
{speaker: 'human', text: '[...] Hey, can you explain what the goEsacpeString function does here?'}
{speaker: 'assistant', text: undefined}
```

5. Send request to Sourcegraph to talk to LLM

With the prompt ready, it's time to send those messages to the LLM. Cody uses the code AI platform's streaming completions[1] API for that. From there the messages are forwarded to the configured LLM model.

Here's what an example request looks like, sent via HTTPS to `'/.api/completions/stream'` on the code AI platform:

[1] So named because the LLMs "complete" a text given to them.

```
Unset
{
  "messages": [
    {
      "speaker": "human",
      "text": "You are Cody, an AI-powered coding assistant created by Sourcegraph. [...]"
    },
    // [...]
    {
      "speaker": "assistant"
    }
  ],
  "temperature": 0.2,
  "maxTokensToSample": 1000,
  "topK": -1,
  "topP": -1
}
```

6. The code AI platform forwards request to LLM

On the code AI platform the request sent by Cody is forwarded to the LLM that was configured by the site administrator.

As of June 8, 2023, that's either Anthropic's Claude or OpenAI's ChatGPT. In the future (see Cody & Sourcegraph's code AI platform in the future), different providers may be available, including self-hosted models.

The LLM replies by streaming its completions back to the code AI platform and from there to Cody.

7. Cody displays LLM reply

Finally, the LLM's response is used by Cody.

If Cody is used as a chatbot, as in the pictured interaction above, then the reply is displayed in the chat window. Since the API is streaming, the user can see the LLM "typing", with the reply seeming to appear syllable by syllable.

If Cody is used in another mode (for example: the user invoked a recipe, or used Cody in inline assist mode, asking it to fix some code), then the LLM's response is further processed. For example: a code snippet that's possibly contained in the LLM's response is extracted and used to replace previously selected text in the user's editor.

8. The user continues the conversation

Even though we've reached the end of the interaction pictured above in the screenshot, it doesn't have to be the end of the conversation between the user and Cody.

The user can ask follow-up questions, for example. In that case, Cody would then include everything that has previously been said – the user's questions and Cody's answers – in the context when answering the last question.

Or the user could reset the conversation, start a new one, use Cody in a different mode, or switch to an older conversation. One question/answer dialogue might just be one of many interactions a user has with Cody throughout their day.

Let's now take a look at the role of Sourcegraph's code AI platform in this interaction.

The unique power of Sourcegraph's code AI platform

The problem of context

Current LLMs, such as GPT-4 or Claude, are trained on vast amounts of data. Some say that they're trained on all of the internet's text, which is probably only a slight exaggeration, since the training data consists of incredible amounts of publicly available data – websites, forums, open source code, comments, social media posts, etc.

But they were not trained on a user's private code. So while LLMs know about open source code and a lot of things about code and programming in general, they don't know about code that's most important to a user – *their* code.

To illustrate, here's me asking Anthropic's Claude about code in the open-source database Redis:

thorsten 13:30
Can you show me a snippet of code from Redis, the in-memory database, to show how it handles strings?

Claude APP 13:30
Here's a snippet from Redis that shows how it handles simple string values:

```
c
/* SET key value
 * SET key value [EX seconds] [PX milliseconds] [NX|XX] */
void setCommand(client *c) {
    int j;
    robj *expire = NULL;
    int unit = UNIT_SECONDS;
    int flags = 0;

    for (j = 3; j < c->argc; j++) {
        char *a = c->argv[j]->ptr;
        robj *next = c->argv[j+1];

        if ((a[0] == 'n' || a[0] == 'N') &&
            (a[1] == 'x' || a[1] == 'X')) {
            flags |= REDIS_SET_NX;
            j++;  // Skip the "NX" argument
        } else if ((a[0] == 'x' || a[0] == 'X') &&
                   (a[1] == 'x' || a[1] == 'X')) {
            flags |= REDIS_SET_XX;
            j++;
        } else if (!strcasecmp(a,"ex")) {
            unit = UNIT_SECONDS;
            expire = next;
            j++;
        } else if (!strcasecmp(a,"px")) {
            unit = UNIT_MILLISECONDS;
            expire = next;
            j++;
        }
    }
```

And indeed, that *is* a snippet of code from Redis that was probably included in the LLMs training data. You can see it here.

If I ask Claude about my own code, which is not open source, it apologizes because it doesn't know about it:

thorsten 13:43
Can you show me a snippet of code from my optimizing compiler `tucan`, hosted at github.com/mrnugget/tucan?

Claude APP 13:43
I apologize, but I do not have access to your private Tucan compiler source code. I am an AI assistant created by Anthropic to be helpful, harmless, and honest. (edited)

But *even if* these LLMs were trained on my code, their knowledge of it would quickly become outdated, since code is constantly being changed. As a matter of fact, that snippet of code from Redis above is nowhere to be found in the current revision of Redis, since it has been refactored away – months after being included in Claude's training data.

Training on a particular codebase would not only be very costly but also worthless after your code has changed.

Another option is to fine-tune an LLM, which requires collecting custom data and using it to re-train the model, making it aware of more data than contained in its initial training data set. That is only slightly more feasible. Collecting, sanitizing, labeling, evaluating training data and re-training an LLM is easier said than done for companies that don't have experience training LLMs.

One could also consider sending their whole codebase to the LLM along with every request. Context windows are becoming bigger and bigger (GPT4's context window allows for 32k tokens, while the newest version of Claude allows for 100k), but codebases today contain millions and millions of lines of code – hard to fit into even the biggest of all context windows. But for argument's sake, let's say the window *is* big enough to contain your whole million line codebase: it would have to be sent along with every request to the LLM – that's slow on one end because the data needs to be transferred over the network(s), expensive because LLM providers charge by used tokens, and slow on the other end because the LLM now has to read a million lines of code before it can reply.

That leaves us with the solution presented in Cheating is All You Need: hand a "cheat sheet" to the LLM that contains just enough (and just the right) information to answer questions about your codebase.

Finding the right context means finding the right code

The context mentioned above, local context and codebase context, is the cheat sheet that Cody hands to the LLM. Putting together the right cheat sheet means finding the right code for a user's question.

Luckily, the Code Search functionality on top of Sourcegraph's code AI platform was built and refined for over a decade to make exactly this easier: finding the right code.

It's Cody's solution to LLMs not knowing about an organization's code: the LLM doesn't need to know about the code. Instead the code AI platform knows the code and allows Cody to find the most relevant code for a given instruction.

Sourcegraph's code AI platform makes Cody universal

Finding the right code within a small- to medium-sized project with 10s of thousands of lines of code (say, expressjs, which has around 15k lines of JavaScript code), might still be possible for a single user to do themselves (using tools such as `ripgrep`, `grep`, `find`, …), but finding the right code inside a company's codebase, across multiple code hosts, in 100s of thousands of lines of code? That's a challenge of a different magnitude, a challenge that Sourcegraph's code AI platform was specifically designed for.

Sourcegraph's code AI platform allows Cody to find code across multiple code hosts, in hundreds of thousands of repositories, in different languages – it makes Cody universal: wherever your code lives, Cody can retrieve it and use it to gather the right context to send to the LLM.

It's not just code either: Cody can use the code AI platform to find documentation, configuration files, and metadata to compile.

In the future, the code AI platform will provide even more and better functionality to Cody that allows it to gather the best possible context for a given user interaction.

Cody & Sourcegraph's code AI platform in the future

Integration with the code graph

As of June 8, 2023, Cody uses Sourcegraph's code AI platform to find (via embeddings search) and retrieve files. The code graph – created with SCIP indexers – is untouched.

By leveraging the code graph, Cody will provide better context to the LLMs. In Why use code graph data we can see how the ability to walk along the code graph allows Cody to not only find files that are semantically related to a user's query but find specific functions, variables, modules and other code structures that are related to a specific bit of code.

This approach matches a sentiment on the future direction of software built on top of LLMs: instead of making the models themselves bigger and bigger, or even training custom models, improve the output from LLMs in a more feasible way by giving the LLM access to tools, such as calculators, code interpreters, or search engines. The code graph provided by the code AI platform is a tool that Cody can use to find more relevant context.

For example, using the code graph Cody could retrieve all the callers of the function a user has a question about. The reply to "How is this function used?" can be improved by looking at how it actually is used, i.e. putting all the call sites into the context that's sent to the LLM.

Using the code graph, Cody could reply to a user's request to find "our most-used React component" by walking the graph and determining which component *is* the most-used one, i.e. which component is actually imported and called the most often in other components.

When the user asks whether a given function needs tests, Cody will be able to check whether that function is called within test files.

Cody's integration with the code AI platform's code graph is coming soon.

Data from external sources

While Big Code is only getting bigger, a lot of possibly relevant context resides far from a company's code and most likely will never be managed in repositories:

- code reviews on code hosts
- tickets in issue trackers
- documentation in wikis
- historical data in version control systems
- performance data and error reports in SaaS applications
- organizational and ownership data in company directories
- internal Q&A boards

All of them contain information that helps developers understand and write code. Soon Cody will be able to access these external data sources and use them to gather relevant context.

A question such as "who owns this code and why?" might be answered by Cody after it looked at the ownership information of the code, its version control history, and consulted the decision log in the company-internal wiki.

If the user asks Cody what the most fragile bit of code in a given module is, it won't just be able to check whether there are enough tests. It will also be able to check with the company's APM tool to find out which line produced the most panics in production.

Context, as we say, is everything. With Sourcegraph's code AI platform, Cody can turn nearly everything into context.

External data plugins will soon be available to Cody.

Cody, everywhere

Today, Cody lives in a user's editor, close to where code is being written, keeping developers in the flow and the inner loop running. It also lives in Sourcegraph's code search UI, which is where developers go to find answers to questions they have about code.

There are many more places though in which someone might have a question about code or needs to understand how a given snippet works or can be extended:

- in code reviews that happen on the code host
- in the browser when looking at documentation
- in the wiki when writing documentation
- in Slack when replying to a colleague
- …

The vision is that Cody will be available in all of these places and more, everywhere a user might need assistance trying to understand, find, or write code.

Since Sourcegraph's code AI platform is already integrated into review tools on code hosts via a browser extension and with more and more data integrations for Cody on the horizon, this is a natural next step.

Improved ranking

As we saw above, one of the key differentiators of Cody is that it builds up the right context for a given user interaction – the context should include files and documents that are relevant to the interaction. Implicit in that is the idea that some files are more relevant than others; there is a ranking. That's where Sourcegraph's decade-long experience building Code Search can help Cody.

How much better will the context that Cody gathers be if we improve how we determine which files are relevant and which aren't? Our bet: a lot better. So far, the embeddings search that Cody uses doesn't take a lot of signals into its ranking. In the future we want to incorporate information from the code graph, from ownership data, from external data and other sources into ranking to return only the truly relevant files to a user's query.

Improved ranking also means we can cut out the files that are not relevant, which shrinks the amount of data that's being sent to the LLM, reducing costs and data egress, and making interactions faster.

Sourcegraph embeddings: no dependency on OpenAI and better-suited models

Today, Sourcegraph's code AI platform creates embeddings using OpenAI's embeddings API. First results of research show that the quality of embeddings can be improved a lot by using a different model specifically selected for Sourcegraph's purpose of creating embeddings for code.

In the future, the code AI platform will be able to create embeddings on its own, without using OpenAI's API and with a custom model, yielding not only better results but also reducing costs for Sourcegraph and its customers. See this section in the Cody white paper on context for more details.